Check 47. Static Taint Analysis Traversal.pdf

by Arta Sundjaja

Submission date: 07-May-2019 05:51PM (UTC+0700)
Submission ID: 1102517599
File name: 47._Static_Taint_Analysis_Traversal.pdf (479.11K)
Word count: 4213
Character count: 24821

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 135 (2018) 596–605

Procedia Computer Science

www.elsevier.com/locate/procedia

3rd International Conference on Computer Science and Computational Intelligence 2018

Static Taint Analysis Traversal with Object Oriented Component for Web File Injection Vulnerability Pattern Detection

Aditya Kurniawan[a,b,*], Bahtiar Saleh Abbas[d,b], Agung Trisetyarso[b], Sani Muhammad Isa[c,b]

[a] *Computer Science Department, School of Computer Science, Bina Nusantara University, Jakarta, Indonesia 11480*

[b] *Computer Science Department, BINUS Graduate Program – Doctor of Computer Science, Bina Nusantara University, Jakarta, Indonesia 11480*

[c] *Computer Science Department, BINUS Graduate Program – Master of Computer Science, Bina Nusantara University, Jakarta, Indonesia 11480*

[d] *Industrial Engineering Department, Faculty of Engineering, Bina Nusantara University, Jakarta, Indonesia 11480*

Abstract

We introduce a composition of object-oriented component PHP grammar for taint analysis. Our novel method successfully restructured the PHP parser and reduced grammar artifact objects that must be visited in a taint analysis process by up to 52% grammar variation. Taint analysis is an analysis that detects any injection vulnerability pattern in source code. The analysis identifies the information flow of untrustworthy input that affects the sensitive sink or part of the system. The static taint analysis was run on an abstract syntax tree and traversed all nodes. A static taint analysis uses a parser to traverse abstract syntax trees of the source code. A web PHP parser has 140 grammar combinations in an abstract syntax tree, which has to be traversed to recognize the tainted flow pattern. Additionally, there are many variations of syntax and coding styles for tainted flow patterns. Therefore, the amount of combinations will consume many computation resources.

Keywords: web shell backdoor; classification; function call

* Corresponding author. Tel.: +6221-5345830 Ext. 2259
E-mail address: adkurniawan@binus.edu

10.1016/j.procs.2018.08.227

1. Introduction

Web backdoor malware has a dangerous capability to fully take over control of a web application and its server host. Web backdoor malware is injected through file injection vulnerabilities in source code. There are three ways it can be injected through vulnerabilities, namely: 1) unrestricted file uploads of dangerous types (code: CWE_434); 2) improper control filenames that include the lor require function (code: CWE_98); 3) improper neutralization of directives in dynamically evaluated code (code: CWE_95). File injection vulnerabilities have various patterns that need to be detected faster and more accurately to ensure safety of the application and server host.

Each programming language has different syntax and grammar processing. This is because every programming language has different parser and lexers. The Turing Machine theory said that every machine only recognized only one language by definition, whether an input is given to machine for acceptance or not[1]. The theory shows that every programming language has its own grammar definition in their parser and lexer. Static code analysis needs the specific parser and lexer to recognize that grammar correctly and give abstract syntax trees as the output to be analyzed. Figure 1 explains how the abstract syntax tree is extracted from PHP source code. The extraction method is using context-free grammar that was defined for that specific programming language.

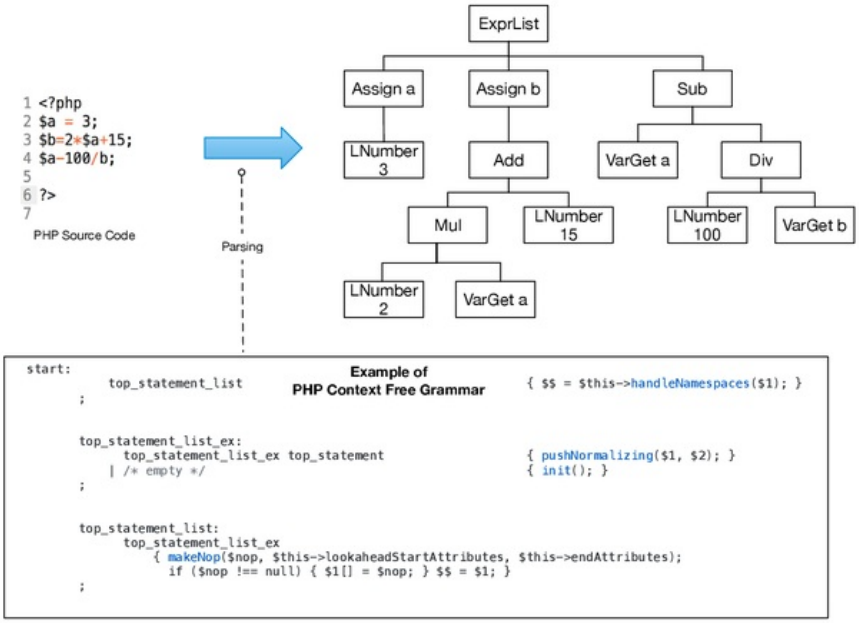

Fig. 1: Abstract Syntax Extraction from Context Free Grammar

Taint analysis uses an abstract syntax tree to traverse the injection vulnerability patterns. The pattern consists of three common steps that relate to each other. The first pattern is a taint input assignment from the first source. The second pattern is a concatenation of assignments or another tainted assignment. The third pattern is an execution of the tainted variable on sensitive functions or the sink part of the source code.

The abstract syntax tree constructed by the parser reads the context-free grammar. Figure 1 shows that the first step is to read the source code and parse it with the grammar. Context-free grammar is defined from the beginning of the program and is a set of recursive rewriting formal rules or grammar used to generate string patterns. The purpose of context-free grammar is to capture all possible token string combinations in a formal language by definition. Therefore, an abstract syntax tree must be used in the taint analysis process.

Table 1: Taint Analysis Research in (2002 - 2017)

Author	Year	Use	Technology
Foster et. al[2]	2002	Buffer Overflow	Linux
Huang et. al[3]	2004	SQL Injection	PHP
Livshits et. al[4]	2005	SQL Injection	Java
Jovanovic et. al[5]	2006	SQL Injection	PHP
Xie and Aiken[6]	2006	SQL Injection	PHP
Conti and Russo[7]	2010	Input Program	Python
Sridharan et. al[8]	2012	Point-to-Analysis	Javascript
Li et. al[9]	2014	Privacy Leak	Java Android
Sampiao and Garcia[10]	2016	Data Flow	Java Eclipse
Schoepe et. al[11]	2016	Shadow Memory	Java Android
Thome et. al[12]	2017	XMLi, XPathi, LDAPi	PHP
Grech and Smaragdakis[13]	2017	Information Flow	Java Android
Wustholz et. al[14]	2017	Denial of Service	Java Servlet

Taint analysis can help identify and locate the vulnerable parts of the source code. A taint analysis identifies malicious input coming from untrusted sources and tracks all variable data affected by the input source. Taint analysis has been widely used throughout the cyber security field. Table 1 shows the implementation of a taint analysis for PHP with various technologies and purposes.

Jovanovic et. al (2006) used context-free grammar to parse and analyze PHP source code for SQL injection and cross-site scripting vulnerability detection (XSS)[5] for the first time. Their method was tested among 44,000 lines of code in SimpGB, a PHP web open source project. Their method needs 185.6 minutes on average to analyze the source code. The method needs 3.9 seconds per line of source code. Both Jovanovic and Fang Yu showed that extracting control flow graphs using context-free grammar is slow. They used context-free grammar that was defined by themselves, and they did not use the native PHP grammar from the PHP Zend engine, so they used context-free grammar to detect those vulnerability patterns as commonly as possible.

Fang Yu et. al (2010)continued JovanovicâĂŹs work by extend their program, Pixy, with an automata string analysis feature[15]. Their method is tested among 44000 lines of code's SimpGB, a PHP web open source. Their method need 185.6 minutes averagely to analyze those source code. The method need 3.9 second/line of source code. Both of Jovanovic and Fang Yu shows that extract control flow graph using context free grammar is slow. They use context free grammar that defined by themselves and not use the native PHP grammar from PHP Zend engine, so they use context free grammar to detect those vulnerability pattern as common as possible.

The abstract syntax tree that uses context-free grammar in native PHP developed by Nikita Popov in 2014. PopovâĂŹs method used PHP YAML (the context-free grammar of PHP) from the PHP Zend Engine. The YAML format is a human-readable context-free grammar that is executed to extract abstract syntax trees from PHP source code.

The abstract syntax tree is created as an object-oriented form. Therefore, the abstract syntax tree nodes will be fully accessible as objects that can affect static analysis quality. The abstract syntax tree parser and lexer library can be accessed at github[1].This library is fully decoupled from the PHP Zend Engine parser to maintain code quality that is compiled by the PHP compiler

PopovâĂŹs PHP abstract syntax tree was developed for the static code analysis field. Hills et al. used this library to analyze and optimize included file features in PHP[16]. They analyzed 4,593,476 lines of code

[1] https://github.com/nikic/PHP-Parser/tree/master

over 20 PHP open source frameworks. Their method needed 5-50 milliseconds per file on average to get their result.

The core idea of our novelty method is to reduce the visited paths of abstract syntax trees by using object-oriented component and interface polymorphism. PopovâĂŹs PHP parser library has 140 nodes of grammar object combinations. Taint analysis does not need to visit every grammar object. Our optimized taint analysis can choose every necessary grammar object that can cause taint operations and analyze the tainted node correlation.

2. Methodology

This section will discuss how our method was constructed. The method has two important elements. The first element is the programming syntax varieties for each programmer. The second element is to restructure the parser grammar into component objects and combine them with the interface polymorphism technique.

2.1. Programming Syntax Varieties

Each programmer has a different style and method that affects their written web code. Events though two programmers can implement same algorithm, but their web code syntax may differ from each other. Table 3 shows a one line statement of code that has many varieties with limitless possibilities. The code is part of the taint analysis syntax category, specifically the tainted source assignment part. For example, *$query="..".$_GET."..." or + with POST*. Lets "..." scalar string be represented by x, dot(.) represented by a, plus(+) represented by b, $_GET represented by c, $_POST represented by d. Tainted source assignment varieties syntax is limitless, since the scalar string concatenation numbers are also limitless. Table 3 shows the possible syntax varieties include $a => c => x => b => x => d$, $a => x => a => d$, etc. Therefore, the most difficult problem of taint analysis is analyzing all possibilities of syntax varieties.

Taint analysis is an information flow analysis that identifies the flow of untrustworthy input that affects any part of the source code. Information flows from source to the destination object, whenever information stored in the source is transferred to a destination object[17].

Malicious inputs are all user input from HTML that is not properly validated or filtered. For example, if the source of variable C value is untrustworthy, then X is tainted, which is symbolized with a tag. Those tags allow the analysis to track transmission of the tainted object along the execution of the program.

Tainted object sources can come from more than two different transmission sources, which is called taint propagation. Let X be an object where the value is transferred to another object Y, upon which Y becomes tainted. Taint analysis syntax varieties need to categorize and generalize patterns of taint analysis part varieties.

Pattern	Syntax Variation	Possible variation	Description
Tainted source assignment	$var = $GET["input"] / POST	$_POST, $_GET	Assignment directly from tainted source
	$query="..".$_GET."..." or + with POST	(dot(.), plus(+)) $_GET , $_POST	Concatenate assignment from tainted source
	exec($var)	$var	Function call Tainted sources
	proc_open($var)	$var	Function call Tainted sources
Concatenate assignment	A1: $query =$taintedVariable / $taintArray[1]	$taintedVar, $taintedArray[1]	assignment directly from tainted variable or array
	$query="..".A1."..." / +	(dot(.), plus(+)) $taintedVar ,$taintedArray[1]	Concatenate assignment from tainted source
	$query="..A1.."	$taintedVar, $taintedArray[1]	Assignment Encapsed String
	$query += "..".A1."..." /.	(dot(.), plus(+)) $taintedVar ,$taintedArray[1]	Concatenate assignment short hand
	$query = $query + "..".A1."..."	(dot(.), plus(+)) $taintedVar ,$taintedArray[1]	Concatenate assignment long hand
Execute on sensitive function	eval(A1)	$taintedVar, $taintedArray[1]	Tainted variable execute on sensitive function
	eval("..A1..")	$taintedVar, $taintedArray[1]	Tainted variable execute on sensitive function
	eval("..".$GET."..") / POST	$_POST, $_GET	Tainted variable execute on sensitive function
	include(A1)	$taintedVar, $taintedArray[1]	Tainted variable execute on sensitive function
	include("..A1..")	$taintedVar, $taintedArray[1]	Tainted variable execute on sensitive function
	include("..".$GET."..") / POST	$_POST, $_GET	Tainted variable execute on sensitive function
	require(A1)	$taintedVar, $taintedArray[1]	Tainted variable execute on sensitive function
	require("..A1..")	$taintedVar, $taintedArray[1]	Tainted variable execute on sensitive function
	require("..".$GET."..") / POST	$_POST, $_GET	Tainted variable execute on sensitive function
	move_uploaded_file(A1)	$taintedVar, $taintedArray[1]	Tainted variable execute on sensitive function
	move_uploaded_file("..A1..")	$taintedVar, $taintedArray[1]	Tainted variable execute on sensitive function
	move_uploaded_file ("..".$GET."..") / POST	$_POST, $_GET	Tainted variable execute on sensitive function

Based on the explanation in Section 1, the taint analysis conducted on file injection vulnerabilities has three main parts. Table 2 show those three parts. *Tainted source assignment* is the first tainted source. These part is the first tainted source. There are four possible syntax variation sources that consist of $_GET or $_POST direct assignment, concatenation of $_GET or $_POST with string, function call *exec* and *proc_open*. The second part is *concatenation of variable assignment*. There are five syntax variation possibilities that consist of tainted variables or array assignments and four string concatenations with tainted sources. The third part is *execution of tainted source on sensitive function*. This final part will detect sensitive sinks when one or more tainted sources transfer into the sensitive functions through the parameters. There are 12 syntax variation possibilities that consist of three variations of *eval*'s function call, three variations of *include*'s function call, three variations of *require*'s function call and three variations of *move_uploaded_file*'s function call.

Table 3: Example of Possibility Varieties of Syntax

	."	dot(.)	plus(+)	$_GET	$_POST
	x	a	b	c	d
Posibility	x=>a=>c=>x=>b=>x=>d				
	c=>a=>x=>a=>d				
	d=>a=>x=>a=>c				
	d=>a=>x=>a=>a				

2.2. Restructured Parser Component with Interface Polymorphism

The PHP Parser Library created by Popov is using a visitor pattern method from the design pattern theory [18]. The visitor pattern method uses 140 grammar objects to traverse abstract syntax trees. The second element of our method is restructuring the parser. The purpose of this element is to use the necessary grammar object for use in the taint analysis. Therefore, our method does not need to use all 140 grammar objects at once.

First, all those grammar objects will be grouped into components artifacts. Each component is created based on their syntax categorization that has a similar behavior or nature. For example, $ConditionalExtractable$ component is a component that has conditional parts, such as $Do\_$, $If\_$, $ElseIf$, $Switch\_$, $Ternary$, and $While$. Every component connected to $VulnerabilityRegexMapper$ class has unique interface that can be seen on Figure 2. The $VulnerabilityRegexMapper$ class is inherited from $NodeVisitorAbstract$. This method will remove unnecessary grammar object that not related with taint analysis.

Every interface has unique methods that have to be implemented in grammar artifact objects as connectors to another object. For example, every artifact object that uses $LeftRightExtractable$ have to implement $ILeftRightExtractable$ interface function. Those functions make object user can traverse child nodes that has left or right parts recursively. Table 4 shows each interface and its description.

Our taint analysis component architecture has the advantage of controlling what paths and nodes have to be traversed based on the taint analysis requirement. Algorithm 1 shows the algorithm will filter grammar artifact objects automatically based on components during the traversal process. Grammar object artifacts will filter automatically based on their interface polymorphism type instance. The algorithm output is an extracted abstract syntax tree of related grammar artifact objects. The taint-related grammar artifact objects are every grammar object that can cause a tainted effect. The traverse algorithm first uses a deep search approach.

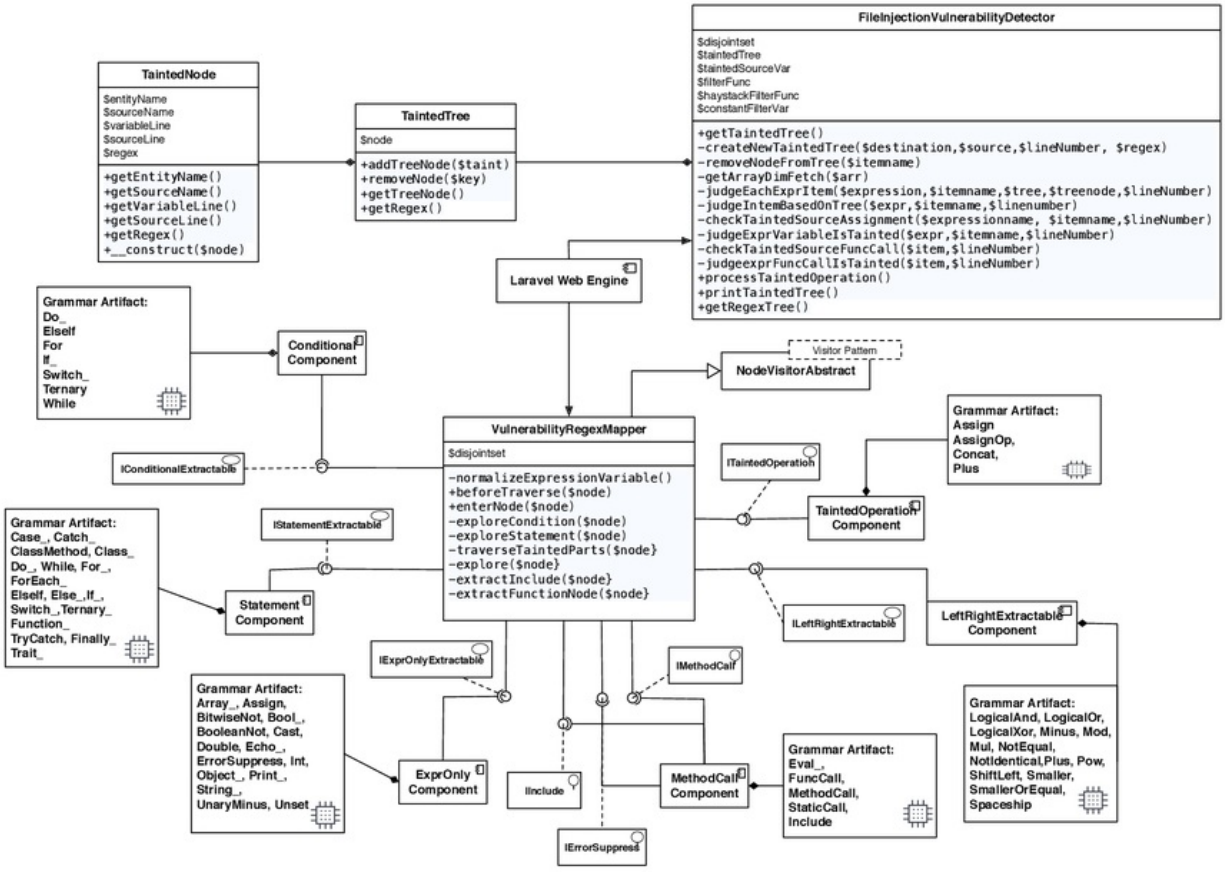

Fig. 2: Architecture of PHP Parser Component Interface for Taint Analysis of File Injection Vulnerability

Table 4: Component Interface of $VulnerabilityRegexMapper$ class Description

Interface	Description	Grammar
IConditionalExtractable	Syntax that has a conditional parts	Do_ ElseIf For If_ Switch_ Ternary While
IStatementExtractable	Syntax that has many statements in curly bracket part	Case_ Catch_ ClassMethod Class_ Do_ ElseIf_ Else_ Finally_ For_ Foreach_ Function_ If_ Switch_ Ternary Trait_ TryCatch While
IExprOnlyExtractable	Syntax that has one expression statement	Array_ Assign BitwiseNot Bool_ BooleanNot Cast Double Echo_ ErrorSuppress Int_ Object_ Print_ String_ UnaryMinus Unset_
ILeftRightExtractable	Syntax that has left or right statement	LogicalAnd,LogicalOr,LogicalXor,Minus, Mod, Mul, NotEqual, NotIdentical, Plus, Pow, ShiftLeft,Smaller,SmallerOrEqual,Spaceship
IMethodCall	Syntax that has method call	Eval_,FuncCall,MethodCall, StaticCall
IInclude	Syntax Include	Include_
IErrrorSupress	Syntax for Error Supress in IMethodCall	Eval_,FuncCall,MethodCall, StaticCall
ITaintedOperation	Syntax assignment that can cause tainted source	Assign,AssignOp,Concat,Plus

Algorithm 1 Traverse Algorithm of VulnerabilityRegexMapper class

```
function TRAVERSEEXPLORE(Node)
    if node instanceof IConditionalExtractable then
        exploreCondition(node)
    end if
    if node instanceof IExprOnlyExtractable then
        exploreStatement(node)
    end if
    if node instanceof ILeftRightExtractable then
        traverseExplore(node.left)
        traverseExplore(node.right)
    end if
    if node instanceof IMethodCall then
        extractFunctionNode(node)
    end if
    if node instanceof IInclude then
        extractInclude(node)
    end if
    if node instanceof IErrorSuppres then
        extractFunctionNode(node)
    end if
     return TaintedGrammar
end function
```

3. Result and Discussion

Table 5: Numbers of Grammar that Used in File Injection Vulnerability Detection

Component	# Numbers of grammar
ConditionalExtractable	7
StatementExtractable	17
ExprOnlyExtractable	15
LeftRightExtractable	14
MethodCall	4
Include	1
ErrrorSupress	4
TaintedOperation	4
Total	**66**

Our algorithm is tested on a Stivalet dataset[19]. Stivalet et. al (2016) generated a large test case for PHP vulnerabilities. . Their dataset was classified based on their category. File injection vulnerability is one of their categorizations. Stivalet et al. categorized file injection vulnerability into the injection category and used the CWE codename. CWE codename, which is related to file injection vulnerability, consists of CWE_95 and CWE_98. Each category is divided by two folders, a safe and unsafe folder. The safe folder contains all files that are grouped as true positives. The unsafe folder contains all files that are grouped as true negatives. There are 672 files in CWE_98 Safe, 25,293 files in CWE_98 Unsafe, 1,296 files in CWE_95 Safe, and 337 files in CWE_95 Unsafe. Code Listing 1 is an example of the Stivalet dataset that is vulnerable to file injection without any filter or validation function.

Code Listing 1: Example of file injection vulnerability source code

```php
<?php
$descriptorspec = array(
0 => array("pipe", "r"),
1 => array("pipe", "w"),
2 => array("file", "/tmp/error-output.txt", "a")
);
$cwd = '/tmp';
$process = proc_open('more /tmp/tainted.txt', $descriptorspec, $pipes, $cwd, NULL);
if (is_resource($process)) {
fclose($pipes[0]);
$tainted = stream_get_contents($pipes[1]);
fclose($pipes[1]);
$return_value = proc_close($process);
}
//$sanitized = filter_var($tainted, FILTER_SANITIZE_MAGIC_QUOTES);
$tainted = $sanitized ;
$query = "'echo $". $tainted . ";'";
$res = eval($query);
?>
```

Based on the architecture in Figure 2, our architecture uses six different components that represent similar grammar artifact objects. Table 5 shows the number of grammar artifact objects for every component. Our file injection vulnerability detection only used 66 grammar artifact object variations from 140 total grammar objects. The result of our novelty method is that we reduced the visited nodes of PHP grammar.

Our method has been tested on those dataset. The implementation of our method executed on hardware that has specification: 1.6 GHz Intel Core i5, 8 GB 1600 MHz DDR3. The result can be seen on Table 6. Table 6 shows at before column is the result before grammar reduction and after column is the result after grammar reduction. Time and memory columns represents the total numbers used for 10 repetition averagely. Our method enhanced 12% of time or reduces 0.591 second, and 7% of memory or reduces 8.225 MB for traverse a tainted tree.

Table 6: Experiment Result

Folder Name	Number of Files	LOC	Before		After	
			Time (s)	**Memory (MB)**	**Time (s)**	**Memory (MB)**
CWE_95 Unsafe	338	21294	1.231	42.5	1.253	39.775
CWE_95 Safe	1298	90860	5.004	130.65	5.066	122.55
CWE_98 Unsafe	672	41664	2.652	70.25	2.358	65.2
CWE_98 Safe	2593	168545	12.76	237.75	10.606	220.725
Average		80590.75	5.41175	120.2875	4.82075	112.0625

4. Conclusion

Programming languages have many grammar artifact object varieties. Every programmer has a different programming syntax and style. Our novel restructured PHP parser method can reduce the amount of grammar objects that must be visited in a taint analysis up to 52% of grammar variation. Our next development is how to combine these architectures into automata generalization for static taint analysis.

References

1. Minsky, M.L.. *Computation: Finite and Infinite Machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.; 1967. ISBN 0-13-165563-9.
2. Foster, J.S., Terauchi, T., Aiken, A.. *Flow-sensitive type qualifiers*; vol. 37. ACM; 2002.
3. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.. Securing web application code by static analysis and runtime protection. In: *Proceedings of the 13th international conference on World Wide Web*. ACM; 2004, p. 40–52.
4. Livshits, V.B., Lam, M.S.. Finding security vulnerabilities in java applications with static analysis. In: *Usenix Security*; vol. 2013. 2005, .
5. Jovanovic, N., Kruegel, C., Kirda, E.. Pixy: A static analysis tool for detecting web application vulnerabilities. In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE; 2006, p. 6–pp.
6. Xie, Y., Aiken, A.. Static detection of security vulnerabilities in scripting languages. In: *USENIX Security*; vol. 6. 2006, p. 179–192.
7. Conti, J.J., Russo, A.. A taint mode for python via a library. In: *Nordic Conference on Secure IT Systems*. Springer; 2010, p. 210–222.
8. Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., Tip, F.. Correlation tracking for points-to analysis of javascript. In: *European Conference on Object-Oriented Programming*. Springer; 2012, p. 435–458.
9. Li, L., Bartel, A., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., et al. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. *arXiv preprint arXiv:14047431* 2014;.
10. Sampaio, L., Garcia, A.. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software* 2016;**113**:337–361.
11. Schoepe, D., Balliu, M., Piessens, F., Sabelfeld, A.. LetâĂŹs face it: Faceted values for taint tracking. In: *European Symposium on Research in Computer Security*. Springer; 2016, p. 561–580.
12. Thome, J., Shar, L.K., Bianculli, D., Briand, L.. Security slicing for auditing common injection vulnerabilities. *Journal of Systems and Software* 2017;.
13. Grech, N., Smaragdakis, Y.. P/taint: unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 2017;**1**(OOPSLA):102.
14. Wüstholz, V., Olivo, O., Heule, M.J., Dillig, I.. Static detection of dos vulnerabilities in programs that use regular expressions (extended version). *arXiv preprint arXiv:170104045* 2017;.
15. Yu, F., Alkhalaf, M., Bultan, T.. Stranger: An automata-based string analysis tool for php. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer; 2010, p. 154–157.
16. Hills, M., Klint, P., Vinju, J.J.. Static, lightweight includes resolution for php. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM; 2014, p. 503–514.
17. Denning, D.E., Denning, P.J.. Certification of programs for secure information flow. *Communications of the ACM* 1977; **20**(7):504–513.
18. Gamma, E.. *Design patterns : elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley; 1995. ISBN 978-0201633610.
19. Stivalet, B., Fong, E.. Large scale generation of complex and faulty php test cases. In: *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE; 2016, p. 409–415.

Check 47. Static Taint Analysis Traversal.pdf

ORIGINALITY REPORT

7%	5%	7%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	stuartgeiger.com Internet Source	3%
2	Michael H Kurniawan, Suharjito, Diana, Gunawan Witjaksono. "Human Anatomy Learning Systems Using Augmented Reality on Mobile Application", Procedia Computer Science, 2018 Publication	2%
3	Gede Putra Kusuma, Evan Kristia Wigati, Yesun Utomo, Louis Khrisna Putera Suryapranata. "Analysis of Gamification Models in Education Using MDA Framework", Procedia Computer Science, 2018 Publication	1%
4	link.springer.com Internet Source	1%

Exclude quotes	On	Exclude matches	< 1%
Exclude bibliography	On		